APPENDIX A

Contents

Section 1 Overview and Pin Functions

1.1 Processor Features

This is a single-chip RISC microprocessor that integrates a RISC-type architecture CPU core and a single precision FPU (Floating Point Unit), an onchip multiplier, a cache memory, and a memory management unit as well as peripheral functions required for system configuration. The processor includes data protection and virtual memory functions.

The Processor also contains the timer, a real time clock, an interrupt controller, and a serial communication interface as peripheral functions necessary for the system configuration. An external memory access support function enables direct connection to DRAM and SDRAM. The Processor microprocessor also supports a PCMCIA interface.

A powerful built-in power management function keeps power consumption low, even during high-speed operation. The Processor can run at four times the frequency of the system bus operating speed, making it optimum for systems requiring both high speed and low power consumption.

The features of the Processor are listed in table 1.1.

Table 1.1 Processor Features

Item	Features
CPU	• Original RISC architecture • 32-bit internal data paths • General-register files — Sixteen 32-bit general registers (eight 32-bit shadow registers) • RISC-type instruction set (upward compatibility with the series) — Instruction length: 16-bit fixed length for improved code efficiency — Load-store architecture — Delayed branch instructions — Instruction set based on C language • Instruction execution time: one cycle for basic instructions • Logical address space: 4 Gbytes (448-Mbytes actual memory space) • Space identifier ASID: 8 bits, 256 logical address spaces • Onchip multiplier • Five-stage pipeline
Operating modes, clock pulse generator	• Operating frequency: 66Mhz (cycle time: 15ns) • Clock mode: selected from an onchip oscillator module, a frequency-doubling circuit, or a clock output by combining them by PLL synchronization • Processing states: — Power-on reset state — Manual reset state — Exception processing state — Program execution state — Power-down state — Bus-released state • Power-down modes: — Sleep mode — Standby mode — Module stop mode • Onchip clock pulse generator

Table 1.1 Processor Features (cont.)

Item	Features
FPU	• Separate Pipeline for Floating Point Operations
	• Single precision floating-point format is supported
	• Subset of IEEE standard's data type is supported
	• Invalid operation and Division by zero exceptions are supported (subset of IEEE standard)
	• Rounding to zero is supported (subsets of IEEE standard)
	• General-register files
	— Sixteen 32-bit floating registers
	• FMAC (multiply & Accumulate) is supported
	• FDIV/FSQRT are supported
	• FLDI0/FLDI1 (load constant0/1) are supported
	• Instruction latency time; two cycles for FMAC/FADD/FSUB/FMUL
	• Execution pitch ; one cycle for FMAC/FADD/FSUB/FMUL

Table 1.1 Processor Features (cont.)

Item	Features
Memory management unit	• 4 Gbytes of address space, 256 address spaces (ASID 8 bits) • Page unit sharing • Supports multiple page sizes: 1, 4 Kbytes • 128-entry, 4-way set associative TLB • Supports software selection of replacement method and random-replacement algorithms • Contents of TLB are directly accessible by address mapping
Cache memory	• 8-Kbytes, unified instruction/data • 128 entries, 4-way set associative, 16-byte block length • Write-back/write-through, LRU replacement algorithm • 1-stage write-back buffer • Contents of cache memory can be accessed directly by address mapping (can be used as onchip memory)
Interrupt controller	• 17 levels of external interrupt (NMI, IRQ), external interrupt pins (NMI, IRL3–IRL0) • Onchip peripheral interrupts: set priority levels for each module • Supports debugging by user break interrupts
User break controller	• 2 break channels • Addresses, data values, type of access, and data size can all be set as break conditions • Supports a sequential break function

Table 1.1 Processor Features (cont.)

Item	Features
Bus state controller	• Supports external memory access — 32/16-bit external data bus • Physical address space divided into seven areas, each a maximum 64 Mbytes, with the following features settable for each area: — Bus size (16 or 32 bits) — Number of wait cycles (also supports a hardware wait function) — Setting the type of space enables direct connection to DRAM, SDRAM, and burst ROM — Supports PCMCIA — Outputs chip select signal (CS0–CS6) for corresponding area • DRAM/SDRAM refresh function — Programmable refresh interval — Supports CAS-before-RAS refresh and self-refresh modes — Supports power-down DRAM • DRAM/SDRAM burst access function • 32bit Multiplex SRAM (address and data multiplexed) burst access function • Switchable between big and little Endian
Timer	• 3-channel auto-reload type 32-bit timer • Input capture function • 6 types of counter input clocks can be selected • Maximum resolution: 2 MHz
Real time clock	• Built-in clock and calendar functions • Onchip 32-kHz crystal oscillator circuit with a maximum resolution (cycle interrupt) of 1/256 second
Serial communication interface	• Select start-stop sync mode or clock sync system • Full-duplex communication • Supports smart card interface
Package	• 144-pin plastic QFP (FP-144A)

1.2 Block Diagram

Figure 1.1 is a functional block diagram of the Processor.

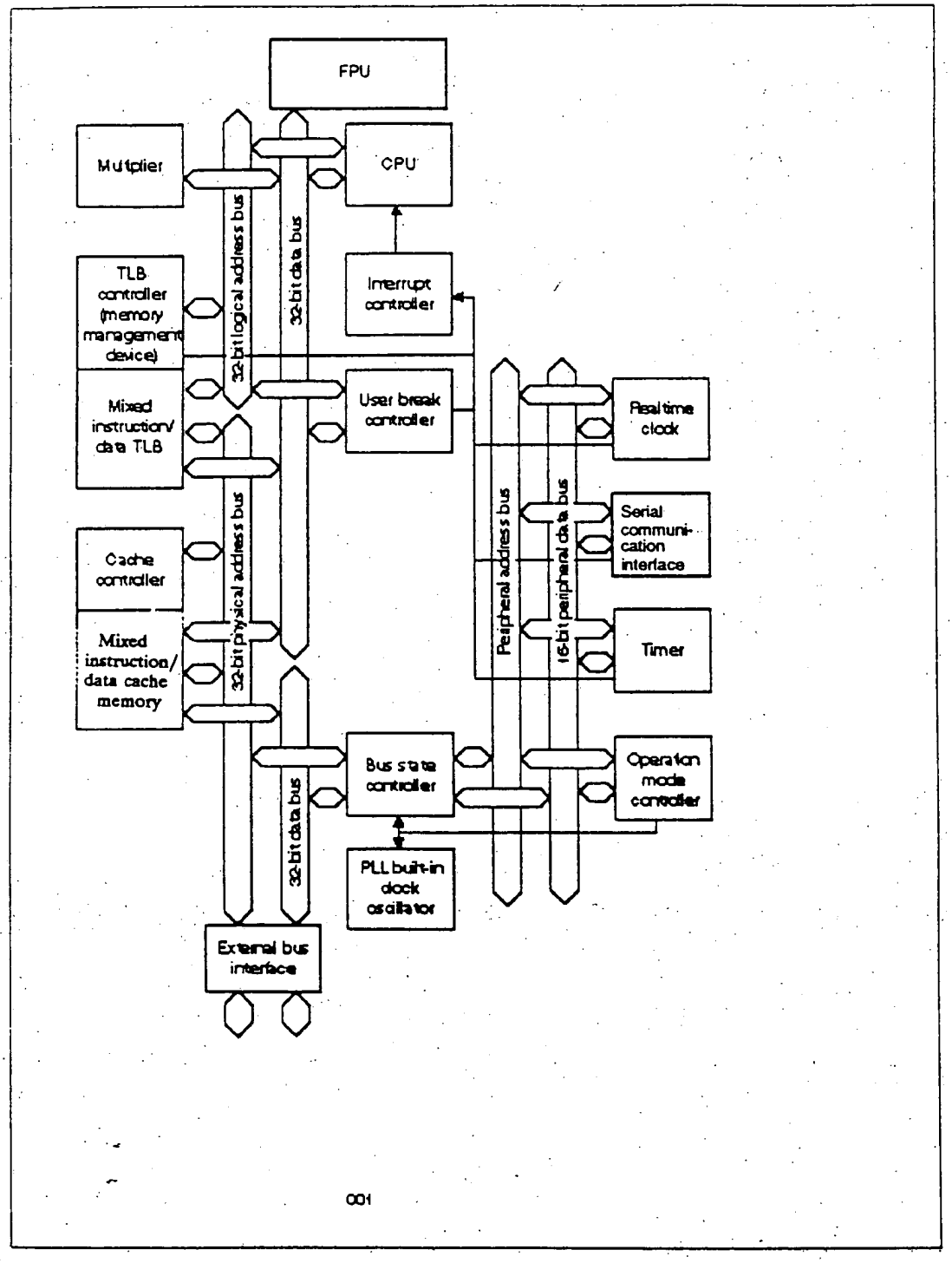


Figure 1.1 Processor Functional Block Diagram

1.3 Pin Description

1.3.1 Pin Arrangement

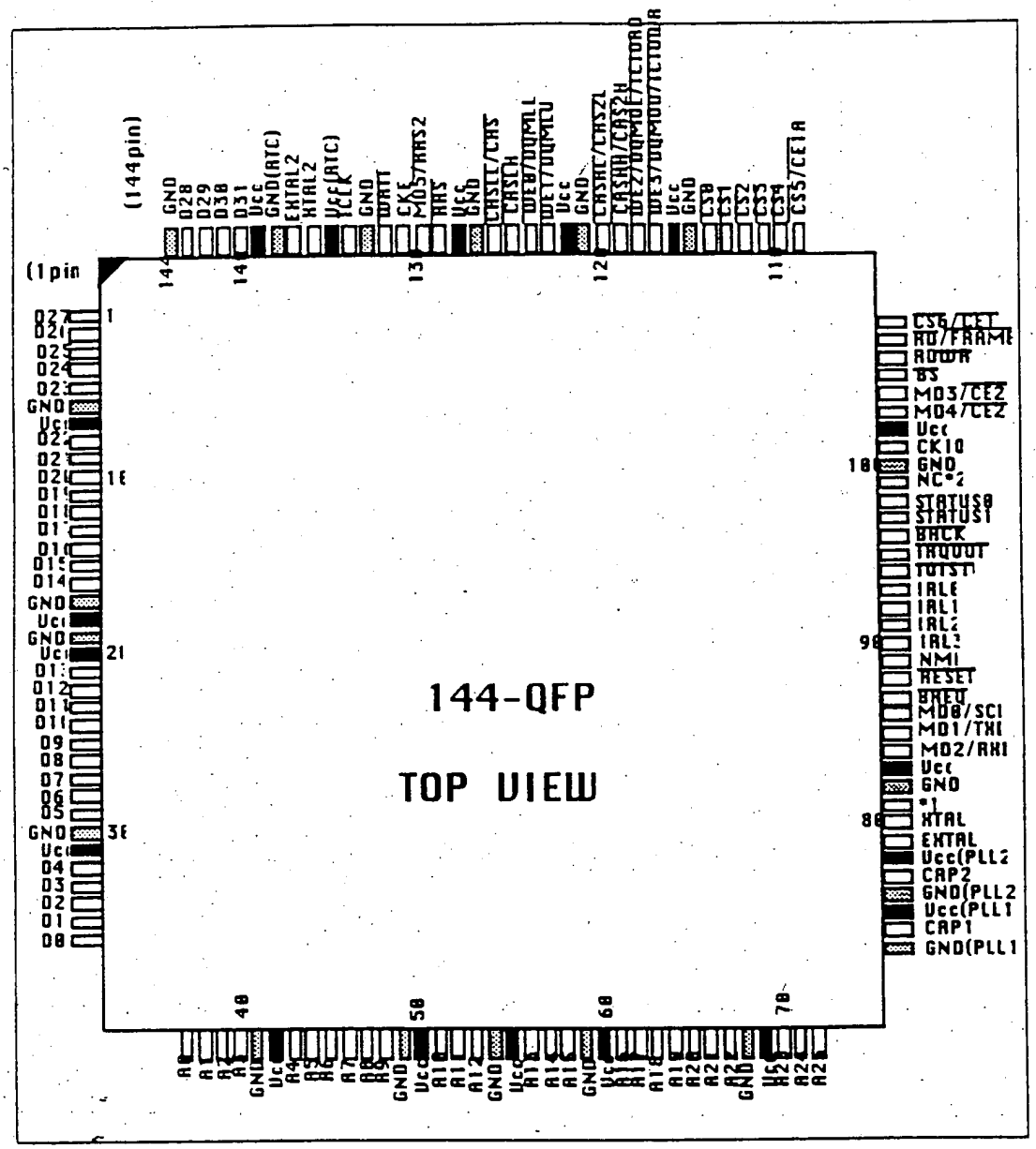


Figure 1.2 Pin Arrangement (144-Pin Plastic QFP)

1.3.2 Processor Pin Functions

Table 1.2 Processor Pin Functions

No.	Terminal	I/O	Description
1	D27	I/O	Data/Address* bus
2	D26	I/O	Data/Address* bus
3	D25	I/O	Data/Address* bus
4	D24	I/O	Data/Address* bus
5	D23	I/O	Data/Address* bus
6	GND	Power	Power (0 V)
7	VCC	Power	Power (3.3 V)
8	D22	I/O	Data/Address* bus
9	D21	I/O	Data/Address* bus
10	D20	I/O	Data/Address* bus
11	D19	I/O	Data/Address* bus
12	D18	I/O	Data/Address* bus
13	D17	I/O	Data/Address* bus
14	D16	I/O	Data/Address* bus
15	D15	I/O	Data/Address* bus
16	D14	I/O	Data/Address* bus
17	GND	Power	Power (0 V)
18	VCC	Power	Power (3.3 V)
19	GND	Power	Power (0 V)
20	VCC	Power	Power (3.3 V)
21	D13	I/O	Data/Address* bus
22	D12	I/O	Data/Address* bus
23	D11	I/O	Data/Address* bus
24	D10	I/O	Data/Address* bus
25	D9	I/O	Data/Address* bus
26	D8	I/O	Data/Address* bus
27	D7	I/O	Data/Address* bus
28	D6	I/O	Data/Address* bus
29	D5	I/O	Data/Address* bus

* notes: Address is available at the MPX-SRAM interface only.

Table 1.2 Processor Pin Functions (cont.)

No.	Terminal	I/O	Description
30	GND	Power	Power (0 V)
31	VCC	Power	Power (3.3 V)
32	D4	I/O	Data/Address* bus
33	D3	I/O	Data/Address* bus
34	D2	I/O	Data/Address* bus
35	D1	I/O	Data/Address* bus
36	D0	I/O	Data/Address* bus
37	A0	O	Address bus
38	A1	O	
39	A2	O	Address bus
40	A3	O	Address bus
41	GND	Power	Power (0 V)
42	VCC	Power	Power (3.3 V)
43	A4	O	Address bus
44	A5	O	Address bus
45	A6	O	Address bus
46	A7	O	Address bus
47	A8	O	Address bus
48	A9	O	Address bus
49	GND	Power	Power (0 V)
50	VCC	Power	Power (3.3 V)
51		O	Address bus
52	A11	O	Address bus
53	A12	O	Address bus
54	GND	Power	Power (0 V)
55	VCC	Power	Power (3.3 V)
56	A13	O	Address bus
57	A14	O	Address bus
58	A15	O	Address bus
59	GND	Power	Power (0 V)
60	VCC	Power	Power (3.3 V)

* notes: Address is available at the MPX-SRAM interface only

Table 1.2 Processor Pin Functions (cont.)

No.	Terminal	I/O	Description
61	A16	O	Address bus
62	A17	O	Address bus
63	A18	O	Address bus
64	A19	O	Address bus
65	A20	O	Address bus
66	A21	O	Address bus
67	A22	O	Address bus
68	GND	Power	Power (0 V)
69	VCC	Power	Power (3.3 V)
70	A23	O	Address bus
71	A24	O	Address bus
72	A25	O	Address bus
73	GND(PLL1)	Power	Power (0 V) for onchip PLL
74	CAP1	O	External capacitance pin for PLL
75	VCC(PLL1)	Power	Power (3.3 V) for onchip PLL
76	GND(PLL2)	Power	Power (0 V) for onchip PLL
77	CAP2	O	External capacitance pin for PLL
78	VCC(PLL2)	Power	Power (3.3 V) for onchip PLL
79	EXTAL	I	External clock/crystal oscillator pin
80	XTAL	O	Crystal oscillator pin
81	—	I	Pull this pin up
82	GND	Power	Power (0 V)
83	VCC	Power	Power (3.3 V)
84	MD2/RXD	I	Operating mode pin/serial data input
85	MD1/TXD	I/O	Operating mode pin/serial data output
86	MD0/SCK	I/O	Operating mode pin/serial clock
87	BREQ	I	Bus request
88	RESET	I	Reset
89	NMI	I	Nonmaskable interrupt request
90	IRL3	I	External interrupt source input
91	IRL2	I	External interrupt source input

Table 1.2 Processor Pin Functions (cont.)

No.	Terminal	I/O	Description
92	IRL1	I	External interrupt source input
93	IRL0	I	External interrupt source input
94	IOIS16	I	IO16-bit instruction
95	IRQOUT	O	Interrupt request notification
96	BACK	O	Bus acknowledge
97	STATUS1	O	Processor status
98	STATUS0	O	Processor status
99	CPACK	O	Clock pause acknowledge
100	GND	Power	Power (0 V)
101	CKIO	I/O	System clock I/O
102	VCC	Power	Power (3.3 V)
103	MD4/CE2B	I/O	Operating mode pin/PCMCIA CE pin
104	MD3/CE2A	I/O	Operating mode pin/PCMCIA CE pin
105	BS	O	Bus cycle start
106	RD/WR	O	Read/write
107	RD/FRAME	O	Read pulse/Frame
108	CS6/CE1B	O	Chip select 6/PCMCIA CE pin
109	CS5/CE1A	O	Chip select 5/PCMCIA CE pin
110	CS4	O	Chip select 4
111	CS3	O	Chip select 3
112	CS2	O	Chip select 2
113	CS1	O	Chip select 1
114	CS0	O	Chip select 0
115	GND	Power	Power (0 V)
116	VCC	Power	Power (3.3 V)
117	WE3/DQMUU/ICIOWR	O	D31–D24 selection signal/IO write
118	WE2/DQMUL/ICIORD	O	D23–D16 selection signal/IO read
119	CASHH/CAS2H	O	D31–D24/D15–D8 selection signal
120	CASHL/CAS2L	O	D23–D16/D7–D0 selection signal
121	GND	Power	Power (0 V)
122	VCC	Power	Power (3.3 V)

Table 1.2 Processor Pin Functions (cont.)

No.	Terminal	I/O	Description
123	WE1/DQMLU	O	D15–D8 selection signal
124	WE0/DQMLL	O	D7–D0 selection signal
125	CASLH	O	D15–D8 selection signal
126	CASLL/CAS	O	D7–D0 selection/memory selection signal
127	GND	Power	Power (0 V)
128	VCC	Power	Power (3.3 V)
129	RAS	O	RAS for DRAM
130	MD5/RAS2	I/O	Operating mode pin/RAS for DRAM
131	CKE	O	Clock enable control for SDRAM
132	WAIT	I	Hardware wait request
133	GND	Power	Power (0 V)
134	TCLK	I/O	Clock I/O for TMU/RTC
135	VCC (RTC)	Power	Power for RTC (3.3 V)
136	XTAL2	O	Crystal oscillator pin for onchip RTC
137	EXTAL2	I	Crystal oscillator pin for onchip RTC
138	GND (RTC)	Power	Power for RTC (0 V)
139	VCC	Power	Power (3.3 V)
140	D31	I/O	Data/Address* bus
141	D30	I/O	Data/Address* bus
142	D29	I/O	Data/Address* bus
143	D28	I/O	Data/Address* bus
144	GND	Power	Power (0 V)

*** notes: Address is available at the MPX-SRAM intetface only**

Section 2 Programming Model of CPU and FPU

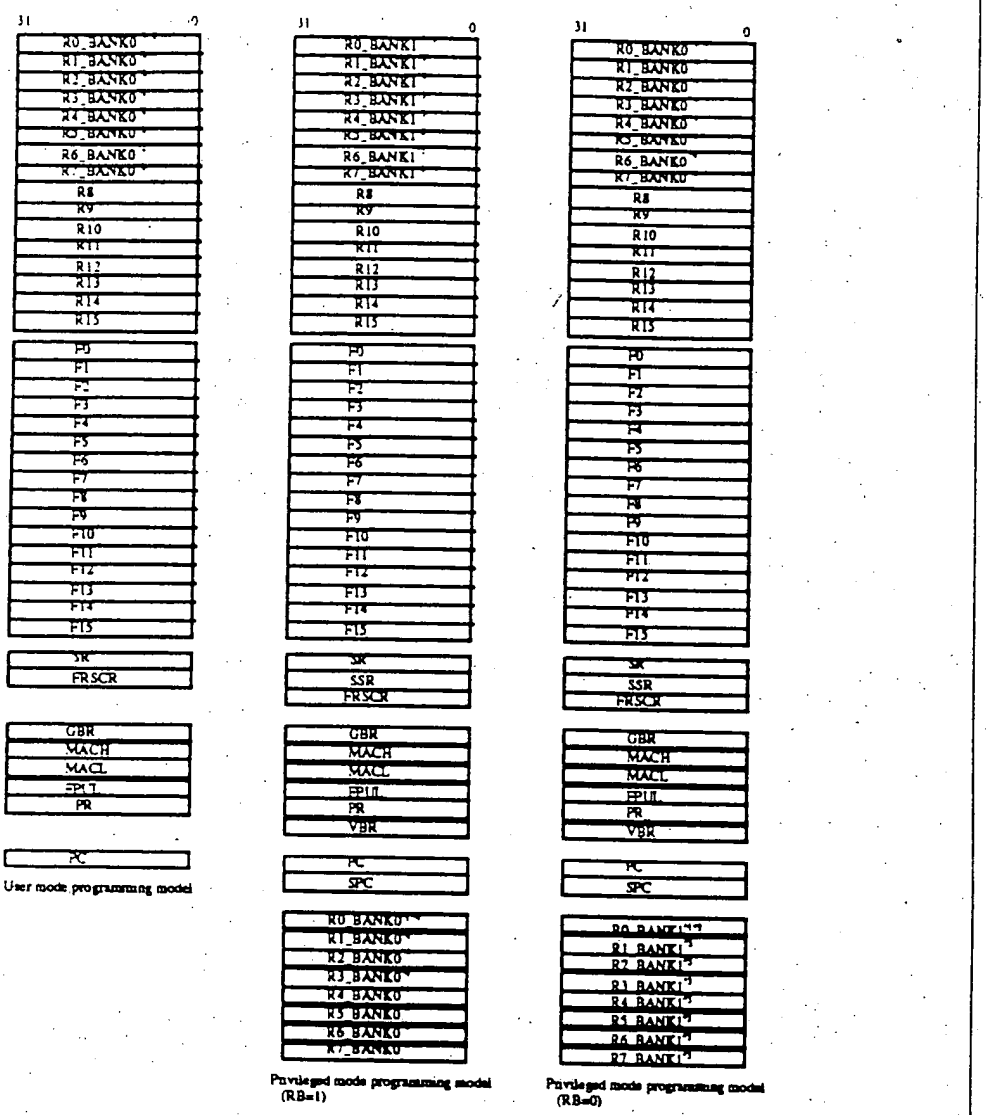
2.1 Programming Model

The Processor operates in user mode under normal conditions and enters privileged mode in response to an exception. Mode is specified by the mode bit (MD) in the status register. The registers accessible to the programmer differ, depending on the processor mode.

General-purpose registers R0 to R7 are banked registers which are switched by a processor mode change. In privileged mode (MD=1), the register bank (RB) bit defines which banked register set is accessed as general-purpose registers, and which set is accessed only through the load control register (LDC) and store control register (STC) instructions.

When the RB bit is a logic one, BANK1 general-purpose registers R0–R7_BANK1 and non-banked general-purpose registers R8–R15 function as the general-purpose register set, with BANK0 general-purpose registers R0–R7_BANK0 accessed only by the LDC/STC instructions.

When the RB bit is a logic zero, BANK0 general-purpose registers R0–R7_BANK0 and non-banked general-purpose registers R8–R15 function as the general-purpose register set, with BANK1 general-purpose registers R0–R7_BANK1 accessed only by the LDC/STC instructions. In user mode (MD=0) BANK0 general purpose registers R0–R7_BANK0 function as the general purpose register set regardless of register bank (RB) setting. The programming model for each processor mode is listed in figure 2.1 and the registers are briefly defined in figures 2.2 and 2.3.



Notes:
1. R0 functions as an index register in the indexed register-indirect addressing mode and indexed GBR-indirect addressing mode. In some instructions, only R0 can be used as the source register or destination register.
2. R0–R7 are banked registers. In user mode, BANK0 is used. In privileged mode, SR.RB specifies BANK. SR.RB = 0: BANK0 is used. SR.RB = 1: BANK1 is used.
3. These registers are only accessed by LDC/STC instructions. SR.RB specifies BANK. SR.RB = 0: BANK0 is used. SR.RB = 1: BANK1 is used.

Figure 2.1 Programming Model of CPU and FPU

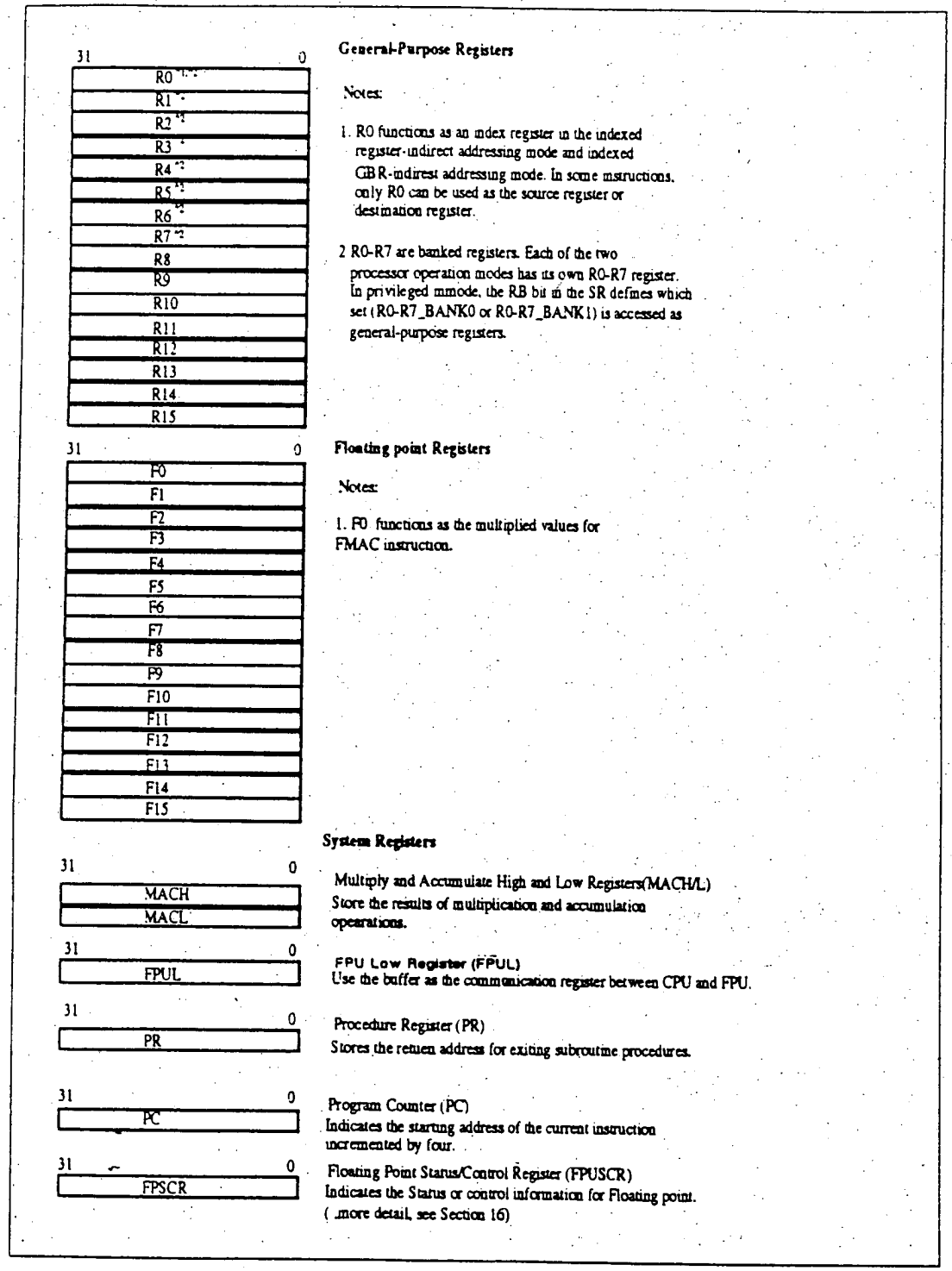
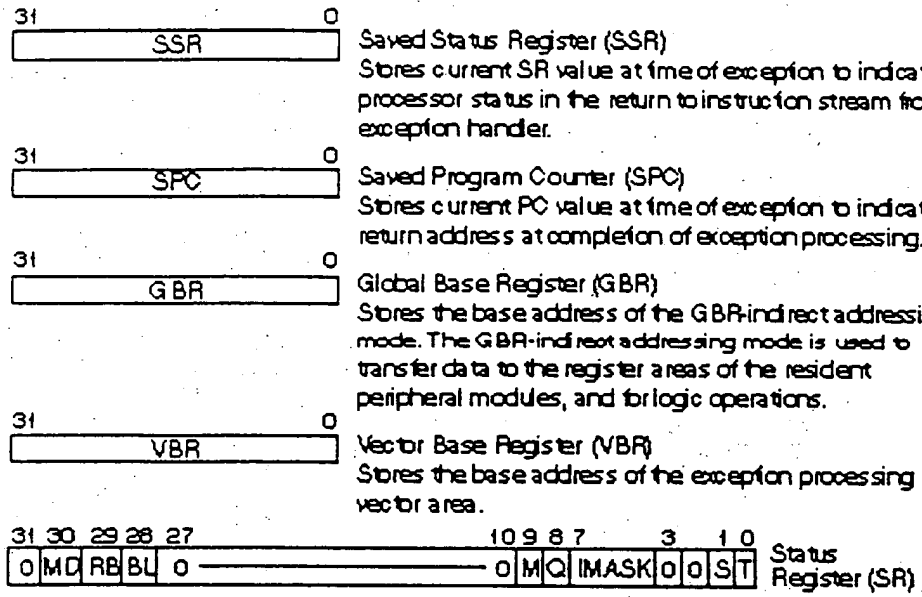


Figure 2.2 Register Set Overview, GPRs, FPRs and System Registers



T bit: The MOVT, CMP/cond, FCMP/cond, TAS, TST, BT, BF, SETT, CLRT, and DT instructions use the T bit to indicate true (logic one) or false (logic zero). The ADDV/C, SUBV/C, DIV0U/S, DIV1, NEGC, SHAR/L, SHLR/L, ROTR/L, and ROTCR/L instructions also use the T bit to indicate a carry, borrow, overflow, or underflow.

S bit: Used by the MAC instruction.

Zero bits: Always read as 0, and should always be written as 0.

IMASK: 4-bit field indicating the interrupt request mask level.

M and Q bits: Used by the DIV0U/S and DIV1 instructions.

RB: Register bank bit: defines the general-purpose registers in privileged mode. A logic one designates R0–R7_BANK1 and R8–R15 are accessed as general-purpose registers, and R0–R7_BANK0 are only accessed by LDC/STC instructions; a logic zero designates R0–R7_BANK0 and R8–R15 are accessed as general-purpose registers, and R0–R7_BANK1 are only accessed by LDC/STC instructions.

BL: Block bit: masks exceptions in privileged mode as follows:
BL = 1, exceptions are masked (not accepted); BL = 0, exceptions are accepted

MD: Processor operation mode bit: indicates the processor operation mode as follows: 1 = Privileged mode; 0 = User mode

Note: Only the M, Q, S, and T bits are read or written from user mode. All other bits are read or written from privileged mode.

Figure 2.3 Register Set Overview, Control Registers

Section 4 Exception Processing

4.1 Overview

Exceptions are deviations from normal program execution that require special handling. The processor responds to an exception by aborting execution of the current instruction (execution is allowed to continue to completion in all interrupt requests) and passing control from the instruction stream to the appropriate user-written exception handling routine.

Usually the contents of PC and SR are saved in the saved program counter (SPC) and saved status register (SSR), respectively, and execution of the exception handler is invoked from a vector location. The return from exception handler (RTE) instruction is issued by the exception handler routine at the completion of the routine, restoring the contents of the PC and SR to recover the instruction stream and the processor status from the point of interruption.

The Processor supports four vector locations. Fixed physical address H'A0000000 is dedicated as a vector for processor resets; other events are assigned offsets within a vector table pointed to by a software-designated vector table base. The types of exception events assigned offsets in the vector table consist of translation lookaside buffer (TLB) miss (H'00000400), general interrupt requests (H'00000600), and general exception events including FPU exception trap other than TLB miss traps (H'00000100). The address of the vector table base is loaded into the vector base register (VBR) by software. The vector table base should reside in P1 or P2 fixed physical address space (figure 4.1).

A basic exception processing sequence consists of the following operations:

- The contents of PC and SR are saved in SPC and SSR, respectively.
- The block (BL) bit in SR is set to a logic one, masking any subsequent exceptions.
- The mode (MD) bit in SR is set to a logic one to place the Processor in privileged mode.
- The register bank (RB) bit in SR is set to a logic one.
- An encoded value identifying the exception event is written into bits 11–0 of the exception event (EXPEVT) or interrupt event (INTEVT) register.
- Instruction execution jumps to the designated vector location to invoke the handler routine.

If a general exception event is detected when the BL bit in SR is a logic one, the event is treated as a reset condition, with execution vectoring to fixed physical address H'A0000000. The SPC and SSR are updated normally, and the code corresponding to the exception event detected is written into the EXPEVT register. If a general interrupt request is detected when BL = 1, the request is masked (held pending) and not accepted until the BL bit is cleared to a logic zero by software.

For reentrant exception processing, SPC and SSR must be saved and the BL bit in SR cleared to a logic zero.

Processor resets and interrupts are asynchronous events unrelated to the instruction stream. All exception events are prioritized to establish an acceptance order whenever two or more exception events occur simultaneously (the power-on reset and manual restart reset are mutually exclusive events). All general exception events occur in a relative order in the execution sequence of an instruction (i.e., execution order), but are handled at priority level 2 in instruction-stream order (i.e., program order), where an exception detected in a preceding instruction is accepted prior to an exception detected in a subsequent instruction (figure 4.1).

Three general exception events (reserved instruction exception, unconditional trap, and illegal slot instruction exception) are detected in the decode stage of different instructions and are mutually exclusive events in the instruction pipeline. In table 5.1, the exception events that trap to a vector location are listed by exception type, instruction completion status, relative priority of acceptance, relative order of occurrence within an instruction execution sequence, and vector location. The exception codes written into bits 11–0 of the EXPEVT register (for reset or general exception events) or the INTEVT register (for general interrupt requests) to identify each specific exception event are defined in table 4.2.

An additional exception register, the TRA register, is used to hold the 8-bit immediate data in an unconditional trap (TRAPA instruction). The bit configurations of the EXPEVT, INTEVT, and TRA registers are diagrammed in figure 4.2.

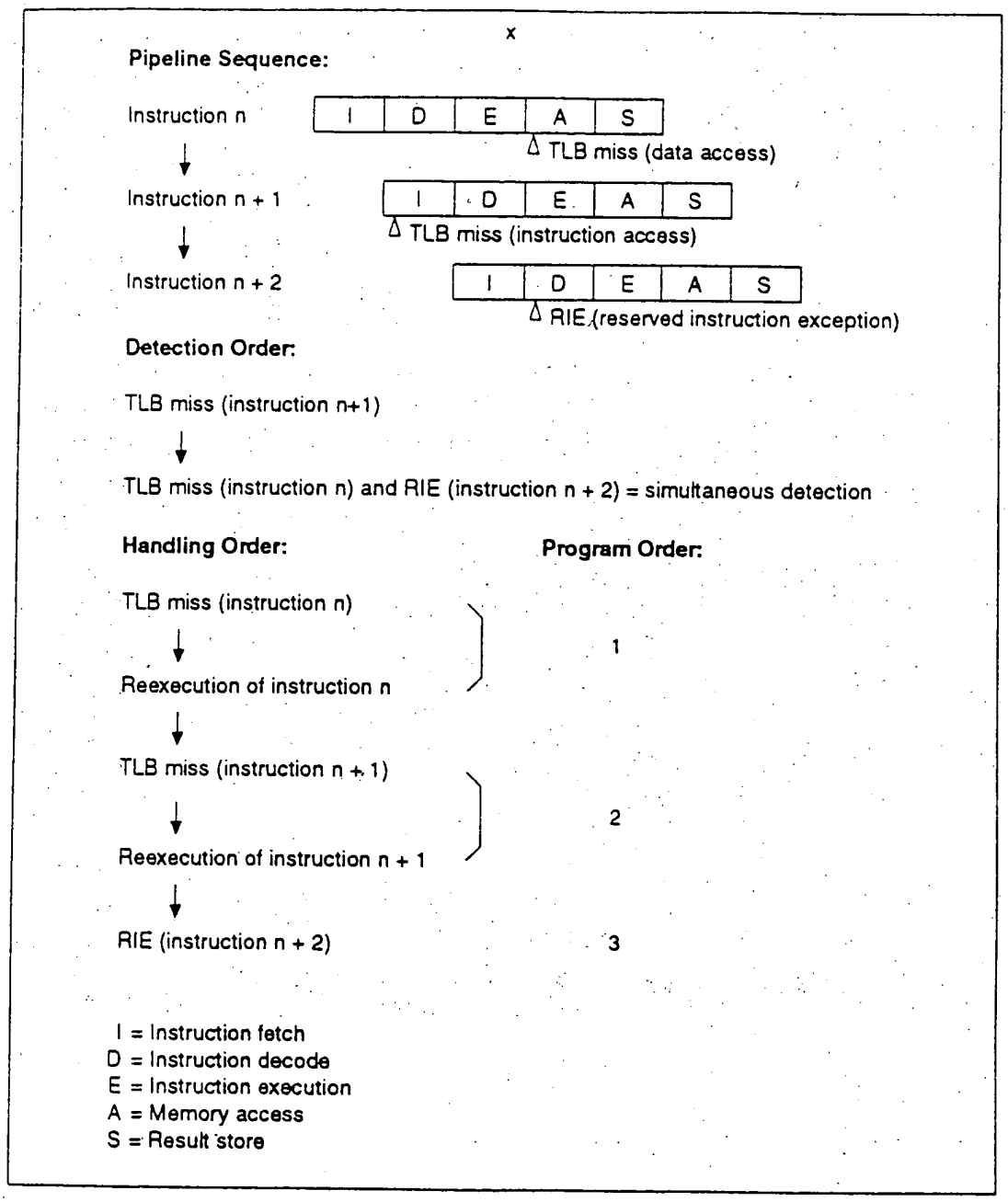


Figure 4.1 Example of Acceptance Order of General Exception Events

Table 4.1 Vectored Exception Events

Exception Type	Current Instruction	Exception Event	Priority	Exception Order	Vector Address	Vector Offset
Reset	Aborted	Power-on	1	—	H'A00000000	—
		Manual reset	1	—	H'A00000000	—
General exception events	Aborted and retried	Address error (instruction access)	2	1	—	H'00000100
		TLB miss (instruction access)	2	2	—	H'00000400
		TLB invalidation (instruction access)	2	3	—	H'00000100
		TLB protection violation (instruction access)	2	4	—	H'00000100
		Reserved instruction exception	2	5	—	H'00000100
		Illegal slot instruction exception	2	5	—	H'00000100
		Address error (data access)	2	6	—	H'00000100
		TLB miss (data access)	2	7	—	H'00000400
		TLB invalidation (data access)	2	8	—	H'00000100
		TLB protection violation (data access)	2	9	—	H'00000100
		FPU exception	2	10	—	H'00000100
		Initial page write	2	11	—	H'00000100
	Completed	Unconditional trap (TRA instruction)	2	5	—	H'00000100
		User breakpoint trap	2	n*2	—	H'00000100
General interrupt requests	Completed	Nonmaskable interrupt	3	—	—	H'00000600
		External hardware interrupt	4*3	—	—	H'00000600
		Peripheral module interrupt	4*3	—	—	H'00000600

Table 4.2 EXPEVT Register and INTEVT Register Exception Codes

Exception Type	Exception Event	Exception Code
Reset	Power-on	H'000
	Manual reset	H'020
General exception events	TLB miss (load)	H'040
	TLB miss (store)	H'060
	Initial page write	H'080
	TLB protection violation (load)	H'0A0
	TLB protection violation (store)	H'0C0
	Address error (load)	H'0E0
	Address error (store)	H'100
	FPU exception	H'120
	Unconditional trap (TRA instruction)	H'160
	Reserved instruction exception	H'180
	Illegal slot instruction exception	H'1A0
	User breakpoint trap	H'1E0

Note: Exception code H'140 is reserved.

Table 4.2 EXPEVT Register and INTEVT Register Exception Codes (cont.)

Exception Type	Exception Event	Exception Code
General interrupt requests	Nonmaskable interrupt	H'1C0
	External hardware interrupt:	
	IRL_0–IRL_3 = 0000	H'200
	IRL_0–IRL_3 = 0001	H'220
	IRL_0–IRL_3 = 0010	H'240
	IRL_0–IRL_3 = 0011	H'260
	IRL_0–IRL_3 = 0100	H'280
	IRL_0–IRL_3 = 0101	H'2A0
	IRL_0–IRL_3 = 0110	H'2C0
	IRL_0–IRL_3 = 0111	H'2E0
	IRL_0–IRL_3 = 1000	H'300
	IRL_0–IRL_3 = 1001	H'320
	IRL_0–IRL_3 = 1010	H'340
	IRL_0–IRL_3 = 1011	H'360
	IRL_0–IRL_3 = 0100	H'380
	IRL_0–IRL_3 = 1101	H'3A0
	IRL_0–IRL_3 = 1110	H'3C0
	IRL_0–IRL_3 = 1111	H'3E0
	Peripheral module interrupts	H'4xx / H'5xx

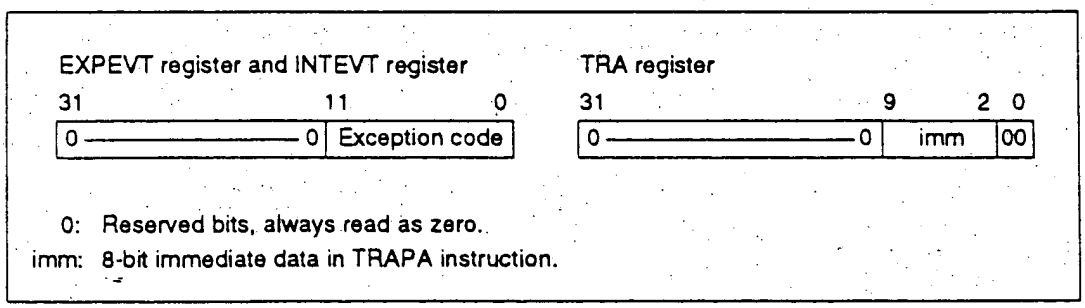


Figure 4.2 Bit Configurations of EXPEVT, INTEVT, and TRA Registers

All interrupts are detected in the decode stage of the instruction pipeline and serviced on instruction boundaries (i.e., the instruction in which the interrupt occurs is allowed to continue to

completion. and upon returning from the interrupt handler, the instruction stream is resumed from the instruction following the instruction in which the interrupt occurred). However, interrupt requests are not accepted in any instruction executed between a delayed branch instruction and the instruction executed in the delay slot.

When a general exception event is detected in a delay slot, the return address saved in SPC is the address of the related delayed branch instruction rather than that of the instruction in which the exception was detected. The illegal slot exception results whenever an attempt is made to execute a TRAPA instruction or certain branch operations in a delay slot. Such branch operations include the JMP, JSR, BRA, BRAF, BSR, BSRF, RTS, RTE, BT, BT/S, BF, and BF/S instructions.

FPU Floating Point Architecture

-------- ISA Specification --------

Contents

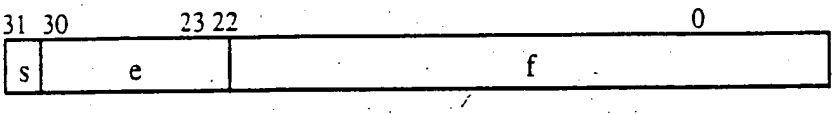
16. Floating Point Architecture

16.1 Introduction

This chapter describes the Instruction Set Architecture of the FPU Floating Point Unit. This architecture has been defined to be compatible with future generation of this architecture. FPU provides a limited set of floating point instructions with the objective of supporting graphics processing in video games. FPU supports single precision floating point operations and FPU will emulate double precision floating point operations to support the full PC API system software. The FPU specification described here is a proper subset of a complete architecture for IEEE 754 floating point architecture.

16.2 Floating Point Format

16.1.1 Floating Point Format



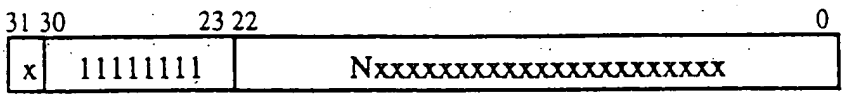
A floating point number contains three fields: a sign, s, an exponent, e, and a fraction, f. The exponent is biased, that is, it is of the form $e = E + bias$. The range of the unbiased exponent E runs from E_{min} -1 to E_{max} +1. Two values are distinguished, E_{min} - 1, which flags zero (both positive and negative) and denormalized numbers, and E_{max} + 1, which flags positive and negative infinity and NaNs (Not a Number). For single precision, the bias is 127, E_{min} is -126 and E_{max} is 127.

The value of a floating point number, v, may be determined as follows:

if $E == E_{max}$ +1 and f != 0, then v is a NaN, regardless of s
if $E == E_{max}$ +1 and f == 0, then $v = (-1)^S$ (infinity)
if $E_{min} <= E <= E_{max}$ then $v = (-1)^S\ 2^E\ (1.f)$
if $E == E_{min}$ -1 and f != 0, then $v = (-1)^S\ 2^{Emin}\ (0.f)$
if $E == E_{min}$ -1 and f == 0, then $v = (-1)^S\ 0$

16.1.2 Not a Number (NaN)

To represent a NaN for a single precision value, at least one of bit 22-0 must be set. Bit 22, if set, indicates a signaling NaN. If reset, the value is a quiet NaN. The bit pattern for NaN is shown in the Figure. The bit N is set for signaling NaNs and reset for quiet NaNs; x indicates a don't-care bit, but at least one of bit 22-0 is set. For NaNs, the sign bit is a don't care.



If the input to the operation generating a floating-point-value is an sNaN:

a. the output is a qNaN if invalid operation exception bit in FPSCR is not enabled;
b. raises invalid operation exception if invalid operation exception bit is enabled in FPSCR. In this case the destination of the operation is not changed.

If the input of the operation generating a floating-point-value is a qNaN and no input of the operation is an sNaN, the output is always a qNaN irrespective of the setting of the invalid operation exception enable bit in FPSCR and will not cause any exception.

For another floating-point operation whose input is a NaN, please refer the individual operation description.

16.1.3 Denormalized Value

A denormalized floating point number is represented with biased exponent to be 0, fraction part to be non-zero and hidden bit is 0. Denormalized numbers (source operand or a result) are uniformly flushed to zero by a value-generating floating-point operation (the operation other than just a copy) in SH3E floating point unit.

16.1.4 Other Special Values

There are several distinguished values in the space of floating point values as shown in Table 1.

Table 1: Representation of Single Precision Special IEEE 754 Values

Value	Representation
+0.0	0x00000000
-0.0	0x80000000
Denormalized Value	(as shown above)
+INF	0x7F800000
-INF	0xFF800000
qNaN, quiet NaN	(as shown above)
sNaN, signaling NaN	(as shown above)

16.3 Floating Point Registers and System Registers for FPU

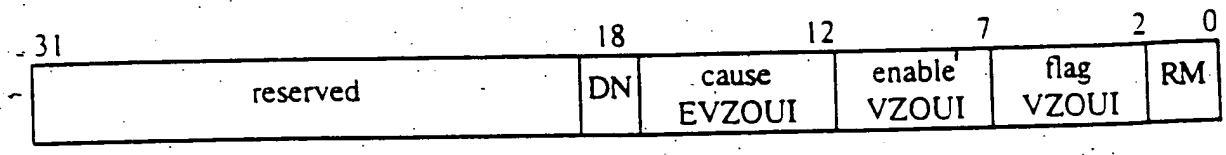
16.3.1 Floating Point Register File

The SH3E provides 16 32-bit single precision floating point registers. Register designators are always 4-bits. In assembly language the floating point registers are designated as FR0, FR1, FR2, and so forth.

16.3.2 Floating Point Communication Register (FPUL)

Information is transferred between the FPU and the CPU through a communication register. FPUL analogous to the MACL and MACH registers of the integer unit. FPUL is considered as a system register, accessed on the CPU side by LDS and STS instructions. FPUL is assigned address 0x03 in the systems register space.

16.3.3 Floating Point Status / Control Register (FPSCR)



The SH3E implements a floating point status and control register, FPSCR, as a system register accessed through the LDS and STS instructions. FPSCR is available for modification by the user program. The FPSCR is part of the per process context and must be saved across context switches and may need to be saved across procedure calls.

The FPSCR is a 32-bit register, which controls the FPU by controlling rounding, gradual underflow (denormalized values), and captures details about floating point exceptions. No FPU enable bit is provided; the FPU is always enabled.

There are five possible FPU exceptions: Inexact (I), Underflow (F), Overflow (O), Division by Zero (Z), and Invalid Operation (V). A sixth exception flag, FPU error (E) is also provided to allow the FPU to report other error conditions. In the FPU, only the V and Z exceptions are supported.

Table 2: Floating Point Exception Flags

Flag	Semantics	Support in this FPU
E	FPU error.	No
V	Invalid Operation.	Yes
Z	Divide By Zero.	Yes
O	Overflow. Value cannot be represented.	No
U	Underflow. Value cannot be represented.	No
I	Inexact. The result cannot be represented.	No

The bits in the **cause** field indicate the cause of exception during the execution of the current instruction. The cause bits are modified by execution of a floating point instruction (capable of causing exception). These bits are set to 0 or 1 depending on occurrence or non-occurrence of exception conditions during the execution of the current instruction. It is possible for the FPU to set more than one bit in this field during the execution of a single instruction, if multiple exception conditions are detected. The bits in the **enable** field indicate the specific type of exceptions that are enabled to raise an exception (i.e., change of flow to an exception handling procedure). An exception is raised if the enable bit and the corresponding cause bit are set by the execution of the current instruction. The bits in the **flag** field are used to capture the cumulative effect of all exceptions during the execution of a sequence of instructions. These bits, once set by an instruction can not be reset by following instructions. The bits in this field can only be reset by an explicit store operation on FPSCR.

In FPU, each bit of cause EOUI, enable OUI, flag OUI and reserved field is predetermined to zero and the following fields are predetermined to the following values :

RM = 01, indicating that rounding is always towards zero; (RZ mode)
DN = 1, indicating that denormalized source or destination operand will be flushed to zero.

The predetermined values cannot be modified even by LDS instruction.

16.4 Floating Point Exception Model

16.4.1 Enabled Exception

Both V and Z exceptions can be enabled by setting its **enable** bit. All exceptions raised by the FPU are mapped onto the same CPU Exception Event. The semantics of the exception are determined by software by reading the system register FPSCR and interpreting the information maintained there.

16.4.2 Disabled Exception

If enable bit V is not set, invalid operations produce qNaN as a result. (except FCMP/FTEST/NAN and FTRC) If enable bit Z is not set, a division by zero produces a correctly signed infinity.

Overflow , underflow , and inexact exceptions do not appear as bit settings in either the **cause** or **flag** fields of the FPSCR. An overflow will produce the number whose absolute value is the largest representable finite number in the format with a

correct sign bit. An underflow will produce a correctly signed zero. If the result of an operation is inexact, the destination register will have the inexact result.

16.4.3 Exception Event and Code for FPU

All FPU exceptions are mapped onto the single general exception event FPU EXCEPTION, which is assigned exception code 0x120. Loads and Stores raise the normal memory management general exceptions.

16.4.4 Alignment of Floating Point Data in Memory

Single precision floating point data is aligned on modulus 4 boundaries, that is, it is aligned in the same fashion as CPU long integers.

16.4.5 Arithmetic with Special Operands conforms to IEEE 754

All arithmetic with special operands (qNaN, sNaN, +INF, -INF, +0, -0) follows IEEE 754 rules. These values are specified for each of the individual operations in this document.

16.5 Synchronization with CPU

16.5.1 Synchronization with CPU

Floating-point operations and CPU operations are issued serially in a program order, but may complete out-of-order, because the execution cycles are different. The floating point operation accessing only FPU resources does not require synchronization with CPU, and the following CPU operations can complete before the completion of the floating point operation. Therefore a proper program can hide the execution cycle of a long execution cycle floating point operation such as Divide. On the other hand, the floating point operation accessing CPU resources such as Compare requires the synchronization to ensure the program order.

16.5.2 Floating Point Operations Requiring Synchronization

Loads, Stores, Compares/test and operations accessing FPUL access CPU resources and require the synchronization. Loads and Stores refer a general register. Post-increment loads and pre-decrement stores modify a general register. Compares/test modify T bit. The operations accessing FPUL refer or modify FPUL. These references and modifications are synchronized with CPU.

16.5.3 Maintaining Program Order on Exceptions

Floating point operations never complete before the following CPU operations complete. FPU EXCEPTION is detected before the following CPU operations complete, and if FPU EXCEPTION is raised, the following operations are canceled. Therefore the program order is ensured even when FPU EXCEPTION is raised.

During a floating point operation execution if a following operation raises an exception, the floating point operation is left executing until FPU resources are accessed, and the access waits the completion of the floating point operation. Therefore the program order is ensured.

16.6 Floating Point Instructions

All floating point instructions are shown in Table 3, and new CPU Instructions Related to FPU are shown in Table 4.

Table 3: Floating Point Instructions

Operation	op code	mnemonic
Floating Move (Load)	FNM8	FMOV.S @Rm, FRn
Floating Move (Store)	FNMA	FMOV.S FRm, @Rn
Floating Move (Restore)	FNM9	FMOV.S @Rm+, FRn
Floating Move (Save)	FNMB	FMOV.S FRm, @-Rn
Floating Move (Load with index)	FNM6	FMOV.S @(R0, Rm), FRn
Floating Move (Store with index)	FNM7	FMOV.S FRm, @(R0, Rn)
Floating Move (in register file)	FNMC	FMOV FRm, FRn
Floating Load Immediate 0	FN8D	FLDI0 FRn
Floating Load Immediate 1	FN9D	FLDI1 FRn
Floating Add	FNM0	FADD FRm, FRn
Floating Subtract	FNM1	FSUB FRm, FRn
Floating Multiply	FNM2	FMUL FRm, FRn
Floating Divide	FNM3	FDIV FRm, FRn
Floating Multiply Accumulate	FNME	FMAC FR0, FRm, FRn
Floating Compare Equal	FNM4	FCMP/EQ FRm, FRn
Floating Compare Greater Than	FNM5	FCMP/GT FRm, FRn
Floating Test NaN	FN7D	FTST/NAN FRn
Floating Negate	FN4D	FNEG FRn
Floating Absolute Value	FN5D	FABS FRn
Floating Square Root	FN6D	FSQRT FRn
Floating Convert from Integer	FN2D	FLOAT FPUL, FRn
Floating Truncate and Convert to Integer	FN3D	FTRC FRm, FPUL
Floating Store from System Register FPUL	FN0D	FSTS FPUL, FRn
Floating Load to System Register FPUL	FN1D	FLDS FRm, FPUL

Table 4: New CPU Instructions Related to FPU

Operation	op code	mnemonic
Load from System Register FPUL	4N5A	LDS Rm, FPUL
Restore System Register FPUL	4N56	LDS.L @Rm+, FPUL
Load from System Register FPSCR	4N6A	LDS Rm, FPSCR
Restore System Register FPSCR	4N66	LDS.L @Rm+, FPSCR
Store to System Register FPUL	0N5A	STS FPUL, Rn
Save System Register FPUL	4N52	STS.L FPUL, @-Rn
Store to System Register FPSCR	0N6A	STS FPSCR, Rn
Save System Register FPSCR	4N62	L FPSCR, @-Rn

16.6.1 Binary Operations

Floating point addition, subtraction, multiplication and division are binary operations following the CPU architectural style.

16.6.2 Comparisons

IEEE 754 mandates four mutually exclusive conditions as the result of a floating point comparison: equality, greater than, less than, and unordered. The approach adopted for floating point compares follows the style of the CPU and provides tests for individual conditions or combinations of conditions. The result of these tests are used to set the T-bit of the CPU and can be used there to control branching or to compute condition related values. Note that FPU supports only FCMP/EQ FRm, FRn , FCMP/GT FRm, FRn, and FTST/NAN FRn . Please look at the Detailed Descriptions how to realize other condition cases respectively.

16.6.3 Loads and Stores

Floating point loads and stores directly access the floating point register file and use the register indirect addressing mode. Post-increment loads and pre-decrement stores are also provided to allow floating point values to be pushed and popped efficiently. And loads with index and stores with index is useful to treat some strucural data more easily. Single precision loads and stores are executed in the same way as long loads/store. They differ only in that the floating point register file is sourced (for a store) and written (for a load).

16.6.4 Other Operations

Most other operations, for example floating point negate, are provided as unary operations to conserve instruction coding space.

16.6.5 CPU Instructions for FPU

LDS and STS instructions are extended to access the new system registers FPUL and FPSCR. The extended part of the instructions will also be described in this section.

16.6.6 Detailed Descriptions

The execution cycles of floating point instructions are defined two values, Latency and Pitch. Latency indicates the cycle to generate the result value, and Pitch indicates the cycle to wait to start the next instruction. Latency and Pitch of almost all CPU instructions are the same, and their execution cycles are expressed by one value.

For most of the operations, we include a special case table to indicate the type of the result for possible source and destination operand types.The operations are described in C language. The common C functions are shown below.

```
#define CAUSE_V    0x00010000  /* FPSCR (bit 16) */
#define CAUSE_Z    0x00008000  /* FPSCR (bit 15) */
#define ENABLE_V   0x00000800  /* FPSCR (bit 11) */
#define ENABLE_Z   0x00000400  /* FPSCR (bit 10) */
#define FLAG_V     0x00000040  /* FPSCR (bit  6) */
#define FLAG_Z     0x00000020  /* FPSCR (bit  5) */

#define NORM       0x0 /* There is no special    */
#define PZERO      0x2 /* meaning for this code  */
#define NZERO      0x3 /* assignment to the data */
#define PINF       0x4 /* types.                 */
#define NINF       0x5 /*                        */
#define sNaN       0x6 /* This is just for       */
#define qNaN       0x7 /* the description.       */
```

```
#define EQ        0x0 /* There is no special    */
#define UO        0x1 /* meaning for this code  */
#define GT        0x2 /* assignment.            */
#define NOTGT     0x3 /* This is just for       */
#define INVALID   0x4 /* the description.       */

long  FPSCR;
int   T;

int load_long(long *adress,*data)
{
      /* This function is defined in CPU part */
}
int store_long(long *adress,*data)
{
      /* This function is defined in CPU part */
}
int sign_of(long *src)
{
      return(*src >> 31);
}
int data_type_of(long *src)
{
float abs;
      abs = *src & 0x7fffffff;
      if(abs < 0x00800000){
        if(sign_of(src) == 0) return(PZERO);
        else                  return(NZERO);
      }
 else if((0x00800000 <= abs) && (abs < 0x7f800000))
                              return(NORM);
 else if(0x7f800000 == abs){
        if(sign_of(src) == 0) return(PINF);
        else                  return(NINF);
      }
 else if(0x00400000 & abs)    return(sNaN);
 else                         return(qNaN);
     }
}
clear_cause_VZ() { FPSCR &= (~CAUSE_V & ~CAUSE_Z); }

set_V() { FPSCR |= (CAUSE_V | FLAG_V); }
set_Z() { FPSCR |= (CAUSE_Z | FLAG_Z); }

invalid(float *dest)
{
  set_V();
  if((FPSCR & ENABLE_V) == 0) qnan(dest);
  }
}
dz(float *dest, int sign)
{
  set_Z();
  if((FPSCR & ENABLE_Z) == 0) inf(dest,sign);
}
zero(float *dest, int sign)
{
      if(sign == 0)     *dest = 0x00000000;
      else              *dest = 0x80000000;
}
inf(float *dest, int sign)
{
```

```
        if(sign == 0)       *dest = 0x7f800000;
        else                *dest = 0xff800000;
}
qnan(float *dest)
{
        *dest = 0x7fbfffff;
}
```

FABS (Floating Point Absolute Value): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FABS FRn	|FRn| -> FRn	1111nnnn01011101	2	1	-

Description: Takes floating-point-arithmetic absolute value of the content of floating point register FRn. And the result of this operation is written on the FRn.

Operation:

```
FABS(float *FRn)  /* FABS FRn */
{
  clear_cause_VZ();
  case(data_type_of(FRn))      {
      NORM :  if(sign_of(FRn) == 0) *FRn =  *FRn;
              else                  *FRn = -*FRn;
                                 break;
      PZERO:
      NZERO:       zero(FRn,0);       break;
      PINF :
      NINF :        inf(FRn,0);       break;
      qNaN :       qnan(FRn);  break;
      sNaN :    invalid(FRn);  break;
  }
  pc += 2;
}
```

FABS Special Cases

FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
FABS (FRn)	ABS	+0	+0	+INF	+INF	qNaN	Invaild

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FADD (Floating Point Add): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FADD FRm, FRn	FRn+FRm -> FRn	1111nnnnmmmm0000	2	1	-

Description: Floating-point-arithmetically adds the contents of floating point registers FRm and FRn. And the result of this operation is written on the FRn.

Operation:

```
FADD(float *FRm,*FRn)    /* FADD FRm,FRn */
{
        clear_cause_VZ();
        if((data_type_of(FRm) == sNaN) ||
           (data_type_of(FRn) == sNaN))  invalid(FRn);
   else if((data_type_of(FRm) == qNaN) ||
           (data_type_of(FRn) == qNaN))  qnan(FRn);
  else case(data_type_of(FRm))      {
     NORM :
       case(data_type_of(FRn))      {
           PINF :       inf(FRn,0);        break;
           NINF :       inf(FRn,1);        break;
         default:       *FRn = *FRn + *FRm;     break;
       }                                  break;
     PZERO:
       case(data_type_of(FRn))      {
           NORM :       *FRn = *FRn + *FRm;     break;
           PZERO:
           NZERO:      zero(FRn,0);        break;
           PINF :       inf(FRn,0);        break;
           NINF :       inf(FRn,1);        break;
       }                                  break;
    NZERO:
       case(data_type_of(FRn))      {
           NORM :       *FRn = *FRn + *FRm;     break;
           PZERO:      zero(FRn,0);        break;
           NZERO:      zero(FRn,1);        break;
           PINF :       inf(FRn,0);        break;
           NINF :       inf(FRn,1);        break;
       }                                  break;
    PINF :
       case(data_type_of(FRn))      {
           NINF :  invalid(FRn);           break;
         default:       inf(FRn,0);        break;
       }                                  break;
    NINF :
       case(data_type_of(FRn))      {
           PINF :  invalid(FRn);           break;
         default:       inf(FRn,1);        break;
       }                                  break;
}
pc += 2;
}
```

FADD Special Cases

FRm \ FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	ADD				-INF		
+0		+0					
-0			-0				
+INF				+INF	Invalid		
-INF	-INF			Invalid	-INF		
qNaN						qNaN	
sNaN							Invalid

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FCMP (Floating Point Compare): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FCMP/EQ FRm, FRn	(FRn==FRm) ? 1:0 ->T	1111nnnnmmmm0100	2	1	1/0
FCMP/GT FRm, FRn	(FRn> FRm) ? 1:0 ->T	1111nnnnmmmm0101	2	1	1/0

Description: Floating-point-arithmetically compares between the contents of floating point registers FRm and FRn. And the result of this operation, true/false, is written on the T bit.

Operation:

```
FCMP_EQ(float *FRm,*FRn)        /* FCMP/EQ FRm,FRn */
{
      clear_cause_VZ();
      if(fcmp_chk(FRm,FRn) == INVALID) {  fcmp_invalid(0); }
      else if(fcmp_chk(FRm,FRn) == EQ)    T = 1;
      else                                T = 0;
      pc += 2;
}
FCMP_GT(float *FRm,*FRn)        /* FCMP/GT FRm,FRn */
{
      clear_cause_VZ();
      if(fcmp_chk(FRm,FRn) == INVALID) {  fcmp_invalid(0); }
      else if(fcmp_chk(FRm,FRn) == GT)    T = 1;
      else                                T = 0;
      pc += 2;
}
fcmp_chk(float *FRm,*FRn)
{
        if((data_type_of(FRm) == sNaN) ||
           (data_type_of(FRn) == sNaN))  return(INVALID);
    else if((data_type_of(FRm) == qNaN) ||
            (data_type_of(FRn) == qNaN))  return(UO);
  else case(data_type_of(FRm))      {
     NORM :  case(data_type_of(FRn))      {
                 PINF : return(GT);              break;
                 NINF : return(NOTGT);           break;
               default:                          break;
             }                                 break;
     PZERO:
     NZERO:  case(data_type_of(FRn))      {
                 PZERO:
                 NZERO: return(EQ);              break;
                 PINF : return(GT);              break;
                 NINF : return(NOTGT);           break;
               default:                          break;
             }                                 break;
     PINF :  case(data_type_of(FRn))      {
                 PINF : return(EQ);              break;
               default: return(NOTGT);           break;
             }                                 break;
     NINF :  case(data_type_of(FRn))      {
                 NINF : return(EQ);              break;
               default: return(GT);              break;
             }                                 break;
  }
  if(*FRn==*FRm)   return(EQ);
  else if(*FRn>*FRm) return(GT);
  else               return(NOTGT);
}
fcmp_invalid(int cmp_flag)
```

```
    set_V();
    if((FPSCR & ENABLE_V) == 0) T = cmp_flag;
}
```

FCMP Special Cases

FRm \ FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	CMP			GT	! GT		
+0		EQ					
-0							
+INF	! GT			EQ			
-INF	GT				EQ		
qNaN						UO	
sNaN							Invalid

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

Note: IEEE defines the independly 4 conditions of caparison. But FPU support FCMP/EQ and FCMP/GT only. But FPU can supprot all conditions using the conbination of BT/BF , FTST/NAN and these 2 FCMPs.

Unorder FRm, FRn	ftst/nan FRm ; bf ; ftst/nan FRn ; bf
(FRm == FRn)	fcmp/eq FRm, FRn ; bt
(FRm != FRn)	fcmp/eq FRm, FRn ; bf
(FRm < FRn)	fcmp/gt FRm, FRn ; bt
(FRm <= FRn)	fcmp/gt FRn, FRm ; bf
(FRm >FRn)	fcmp/gt FRn, FRm ; bt
(FRm >= FRn)	fcmp/gt FRm, FRn ; bf

FDIV (Floating Point Divide): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FDIV FRm, FRn	FRn/FRm -> FRn	1111nnnnmmmm0011	13	12	-

Description: Floating-point-arithmetically divides the content of floating point register FRn by the content of floating point register FRm. And the result of this operation is written on the FRn.

Operation:

```
FDIV(float *FRm,*_FRn)   /* FDIV FRm,_FRn */
{
          clear_cause_VZ();
          if((data_type_of(FRm) == sNaN) ||
             (data_type_of(FRn) == sNaN))  invalid(FRn);
     else if((data_type_of(FRm) == qNaN) ||
             (data_type_of(FRn) == qNaN))  qnan(FRn);
    else case(data_type_of(FRm))      {
       NORM :
         case(data_type_of(FRn))      {
             PINF :
             NINF :  inf(FRn,sign_of(FRm)^sign_of(FRn));      break;
           default:  *FRn = *FRn / *FRm;                      break;
         }                                                    break;
       PZERO:
       NZERO:
         case(data_type_of(FRn))      {
             PZERO:
             NZERO:  invalid(FRn);                            break;
           default:  dz(FRn,sign_of(FRm)^sign_of(FRn));       break;
         }                                                    break;
       PINF :
       NINF :
         case(data_type_of(FRn))      {
             PINF :
             NINF :  invalid(FRn);                            break;
           default:  zero(FRn,sign_of(FRm)^sign_of(FRn));     break;
         }                                                    break;
   }
   pc += 2;
}
```

FDIV Special Cases

FRm \ FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	DIV	0		INF			
+0	DZ	Invalid		DZ			
-0							
+INF	0	+0	-0	Invalid			
-INF		-0	+0				
qNaN	qNaN					qNaN	
sNaN	Invalid						Invalid

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FLDI0 (Floating Point Load Immediate 0): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FLDI0 FRn	0x00000000 -> FRn	1111nnnn10001101	2	1	-

Description: Loads floating point zero (0x00000000) to the floating point register FRn.

Operation:

```
FLDI0(float *FRn) /* FLDI0 FRn */
{
      *FRn  = 0x00000000;
       pc += 2;
}
```

Exceptions:
None

FLDI1 (Floating Point Load Immediate 1): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FLDI1 FRn	0x3F800000 -> FRn	1111nnnn10011101	2	1	-

Description: Loads floating point one (0x3F800000) to the floating point register FRn.

Operation:

```
FLDI1(float *FRn) /* FLDI1 FRn */
{
      *FRn  = 0x3F800000;
       pc += 2;
}
```

Exceptions:
None

FLDS (Floating Point Load to System Register): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FLDS FRm,FPUL	FRm -> FPUL	1111nnnn00011101	2	1	-

Description: Copies the content of floating point register FRm to the system register FPUL.

Operation:

```
FLDS(float *FRm,*FPUL)          /* FLDS FRm,FPUL */
{
        *FPUL = *FRm;
         pc   += 2;
}
```

Exceptions:
None

FMUL (Floating Point Multiply): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FMUL FRm, FRn	FRn * FRm -> FRn	1111nnnnmmmm0010	2	1	-

Description: Floating-point-arithmetically multiplies the contents of floating point registers FRm and FRn. And the result of this operation is written on the FRn.

Operation:

```
FMUL(float *FRm,*FRn)  /* FMUL FRm,FRn */
{
          clear_cause_VZ();
          if((data_type_of(FRm) == sNaN) ||
             (data_type_of(FRn) == sNaN))  invalid(FRn);
     else if((data_type_of(FRm) == qNaN) ||
             (data_type_of(FRn) == qNaN))  qnan(FRn);
   else case(data_type_of(FRm))     {
       NORM :
         case(data_type_of(FRn))     {
             PINF :
             NINF :  inf(FRn,sign_of(FRm)^sign_of(FRn));      break;
           default:  *FRn = (*FRn) * (*FRm);                  break;
         }                                    break;
       PZERO:
       NZERO:
         case(data_type_of(FRn))     {
             PINF :
             NINF :  invalid(FRn);                            break;
           default:  zero(FRn,sign_of(FRm)^sign_of(FRn));     break;
         }                                    break;
       PINF :
       NINF :
         case(data_type_of(FRn))     {
             PZERO:
             NZERO:  invalid(FRn);                            break;
           default:  inf(FRn,sign_of(FRm)^sign_of(FRn));      break;
         }                                    break;
  }
  pc += 2;
}
```

FMUL Special Cases

FRm \ FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	MUL	0		INF			
+0	0	+0	-0	Invalid			
-0		-0	+0				
+INF	INF	Invalid		+INF	-INF		
-INF				-INF	+INF		
qNaN	qNaN					qNaN	
sNaN	Invalid						

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FNEG (Floating Point Negate): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FNEG FRn	-FRn -> FRn	1111nnnn01001101	2	1	-

Description: Floating-point-arithmetically negates the content of floating point register FRn. And the result of this operation is written on the FRn.

Operation:

```
FNEG(float *FRn)  /* FNEG FRn */
{
  clear_cause_VZ();
  case(data_type_of(FRn))      {
      qNaN :       qnan(FRn);  break;
      sNaN :    invalid(FRn);  break;
    default:    *FRn = -(*FRm);       break;
  }
  pc += 2;
}
```

FNEG Special Cases

FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
FNEG (F_n)	NEG	-0	+0	-INF	+INF	qNaN	Invaild

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FSQRT (Floating Square Root): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FSQRT FRn	√FRn -> FRn	1111nnnn01101101	13	12	-

Description: Takes floating-point-arithmetic square root of the content of floating point register FRn. And the result of this operation is written on the FRn.

Operation:

```
FSQRT(float *FRn) /* FSQRT FRn */
{
  clear_cause_VZ();
  case(data_type_of(FRn))      {
      NORM :  if(sign_of(FRn)==0)
                          *FRn = sqrt(*FRn);
              else        invalid(FRn);      break;
      PZERO:
      NZERO:
      PINF :    *FRn = *FRn ;                break;
      NINF :   invalid(FRn);                 break;
      qNaN :      qnan(FRn);                 break;
      sNaN :   invalid(FRn);                 break;
  }
  pc += 2;
}
```

FSQRT Special Cases

FRn	+NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
FSQRT (FRn)	SQRT	Invaild	+0	-0	+INF	Invaild	qNaN	Invaild

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FSTS (Floating Point Store from System Register): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FSTS FPUL,FRn	FPUL -> FRn	1111nnnn00001101	2	1	-

Description: Copies the content of the system register FPUL to floating point register FRn.

Operation:

```
FSTS(float *FRn,*FPUL)   /* FSTS FPUL,FRn */
{
      *FRn   = *FPUL;
      pc   += 4;
}
```

Exceptions:
None

FSUB (Floating Point Subtract): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FSUB FRm, FRn	FRn - FRm -> FRn	1111nnnnmmmm0001	2	1	-

Description: Floating-point-arithmetically subtracts the content of floating point register FRm from the content of floating point register FRn. And the result of this operation is written on the FRn.

Operation:

```
FSUB(float *FRm,*FRn)   /* FSUB FRm,FRn */
{
         clear_cause_VZ();
         if((data_type_of(FRm) == sNaN) ||
            (data_type_of(FRn) == sNaN))  invalid(FRn);
    else if((data_type_of(FRm) == qNaN) ||
            (data_type_of(FRn) == qNaN))  qnan(FRn);
   else case(data_type_of(FRm))      {
      NORM :                          /
        case(data_type_of(FRn))       {
            PINF :       inf(FRn,0);        break;
            NINF :       inf(FRn,1);        break;
          default:  *FRn = *FRn - *FRm;     break;
        }                             break;
      PZERO:
        case(data_type_of(FRn))       {
            NORM :  *FRn = *FRn - *FRm;     break;
            PZERO:     zero(FRn,0);         break;
            NZERO:     zero(FRn,1);         break;
            PINF :      inf(FRn,0);         break;
            NINF :      inf(FRn,1);         break;
        }                             break;
      NZERO:
        case(data_type_of(FRn))       {
            NORM :  *FRn = *FRn - *FRm;     break;
            PZERO:
            NZERO:     zero(FRn,0);         break;
            PINF :      inf(FRn,0);         break;
            NINF :      inf(FRn,1);         break;
        }                             break;
      PINF :
        case(data_type_of(FRn))       {
            NINF :  invalid(FRn);           break;
          default:      inf(FRn,1);         break;
        }                             break;
      NINF :
        case(data_type_of(FRn))       {
            PINF :  invalid(FRn);           break;
          default:      inf(FRn,0);         break;
        }                             break;
  }
  pc += 2;
}
```

FSUB Special Cases

fm \ fn	NORM	-0	+0	+INF	-INF	qNaN	sNaN
NORM	SUB			+INF	-INF		
-0			+0				
+0		-0					
-INF	-INF			Invalid			
+INF	+INF				Invalid		
qNaN						qNaN	
sNaN							Invalid

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FTRC (Floating Point Truncate and Convert to Integer): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FTRC FRm, FPUL	(long)FRm -> FPUL	1111nnnn00111101	2	1	-

Description: Interprets the content of floating point register FRm as a floating point number value and truncates it to an integer value. And the result of this operation is written to the FPUL.

Operation:

```
#define N_INT_RENGE 0xc7000000 /* -1.000000 * 2^31 */
#define P_INT_RENGE 0x46ffffff /*  1.fffffe * 2^30 */

FTRC(float *FRm, int *FPUL)   /* FTRC FRm,FPUL */
{
  clear_cause_VZ();
  case(ftrc_type_of(FRm){
      NORM :      *FPUL = (long)(*FRm);   break;
      PINF :      ftrc_invalid(0);        break;
      NINF :      ftrc_invalid(1);        break;
  }
  pc += 2;
}
int ftrc_type_of(long *src)
{
long  abs;
      abs = *src & 0x7fffffff;
  if(sign_of(src) == 0){
        if(abs > 0x7f800000 ) return(NINF); /* NaN                  */
    else if(abs > P_INT_RENGE) return(PINF); /* out of renge,+INF */
    else                       return(NORM); /* +0,+NORM          */
  }
  else {
      if(abs > N_INT_RENGE)    return(NINF); /* out of renge,+INF,NaN */
      else                     return(NORM); /* -0,-NORM              */
  }
}
ftrc_invalid(long *dest, int sign)
{
  set_V();
  if((FPSCR & ENABLE_V) == 0){
      if(sign == 0)     *dest = 0x7fffffff;
      else              *dest = 0x80000000;
  }
}
```

FTRC Special Cases

FRn	NORM	+0	-0	positive out of range	negaitive out of range	+INF	-INF	qNaN	sNaN
FTRC (FRn)	TRC	0	0	Invaild +MAX	Invaild -MAX	Invaild +MAX	Invaild -MAX	Invaild -MAX	Invaild -MAX

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FTST (Floating Point Test): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FTST/NAN FRn	(FRn==NAN)? 1:0 ->T	1111nnnn01111101	2	1	1/0

Description: Floating-point-arithmetically tests which the contents of floating point register FRn is NAN or not. And the result of this operation, true/false, is written on the T bit.

Operation:

```
FTST_NAN(float *FRn)     /* FTST/NAN FRn */
{
      clear_cause_VZ();
      case(data_type_of(FRn)) {
      NORM :
      PZERO:
      NZERO:
      PINF :
      NINF :        T = 0;      break;
      qNaN :        T = 1;      break;
      sNaN :        fcmp_invalid(1); break;
  }
      pc += 2;
}
```

FTST/NAN Special Cases

FRn	NORM	+0	-0	+INF	-INF	qNaN	sNaN
FTST/NAN (FRn)	T=0	T=0	T=0	T=0	T=0	T=1	Invaild

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

LDS (Load to System Register): CPU Instruction

	Format	Abstract	Code	Latency	Pitch	T bit
1	LDS Rm ,FPUL	Rm->FPUL	0100nnnn01011010	1	1	-
2	LDS.L @Rm+,FPUL	@Rm->FPUL ,Rm+=4	0100nnnn01010110	2	1	-
3	LDS Rm ,FPSCR	Rm->FPSCR	0100nnnn01101010	1	1	-
4	LDS.L @Rm+,FPSCR	@Rm->FPSCR,Rm+=4	0100nnnn01100110	2	1	-

Description:

1. Copies the content of general purpose register Rm to system register FPUL.
2. Loads the content of the memory location addressed by general register Rm. And the result of this operation is written on system register FPUL. Upon successful completion, the value in Rm is incremented by 4.
3. Copies the content of general purpose register Rm to system register FPSCR. The predetermined bits of FPSCR remain unchanged.
4. Loads the content of the memory location addressed by general register Rm. And the result of this operation is written on system register FPSCR. Upon successful completion, the value in Rm is incremented by 4. The predetermined bits of FPSCR remain unchanged.

Operation:

```
#define FPSCR_MASK 0x00018c60

LDS(long  *Rm,*FPUL)            /* LDS Rm,FPUL */
{
      *FPUL = *Rm;
       pc  += 2;
}
LDS_RESTORE(long  *Rm,*FPUL)          /* LDS.L @Rm+,FPUL */
{
      if(load_long(Rm,FPUL) != Address_Error)    *Rm += 4;
       pc += 2;
}
LDS(long  *Rm,*FPSCR)           /* LDS Rm,FPSCR */
{
      *FPSCR = *Rm & FPSCR_MASK;
       pc   += 2;
}
LDS_RESTORE(long  *Rm,*FPSCR)         /* LDS.L @Rm+,FPSCR */
{
long  *tmp_FPSCR;
      if(load_long(Rm,tmp_FPSCR) != Address_Error){
            *FPSCR = *tmp_FPSCR & FPSCR_MASK;
            *Rm += 4;
      }
       pc += 2;
}
```

Exceptions:

Address Error

STS (Store from System Register): CPU Instruction

	Format	Abstract	Code	Latency	Pitch	T bit
1	STS FPUL , Rn	FPUL -> Rn	0100nnnn01011010	1	1	-
2	STS.L FPUL ,@-Rn	Rn-=4,FPUL ->@Rn	0100nnnn01010110	2	1	-
3	STS FPSCR, Rn	FPSCR-> Rn	0100nnnn01101010	1	1	-
4	STS.L FPSCR,@-Rn	Rn-=4,FPSCR->@Rn	0100nnnn01100110	2	1	-

Description:

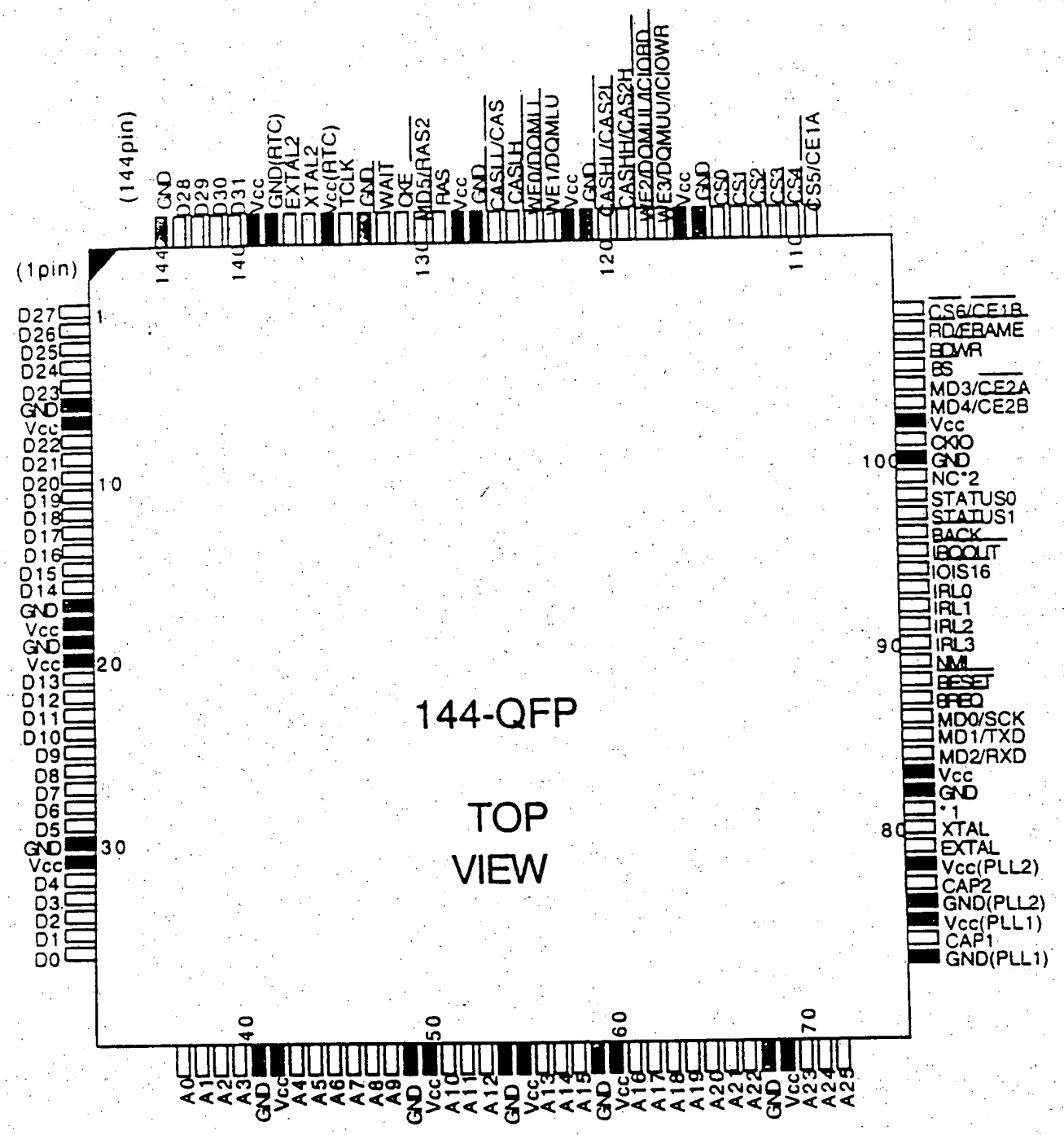
1. Copies the content of system register FPUL to general purpose register Rm.
2. Stores the content of system register FPUL into the memory location addressed by general register Rn decremented by 4. Upon successful completion, the decremented value becomes the value of Rn.
3. Copies the content of system register FPSCR to general purpose register Rm.
4. Stores the content of system register FPSCR into the memory location addressed by general register Rn decremented by 4. Upon successful completion, the decremented value becomes the value of Rn.

Operation:

```
STS(long  *FPUL,*Rn)              /* STS FPUL,Rn */
{
      *Rn  = *FPUL;
       pc += 2;
}
STS_SAVE(long  *FPUL,*Rn)              /* STS.L FPUL,@-Rn */
{
long  *tmp_address = *Rn - 4;
      if(store_long(FPUL,tmp_address) != Address_Error)
      Rn  = tmp_address;
      pc += 2;
}
STS(long  *FPSCR,*Rn)             /* STS FPSCR,Rn */
{
      *Rn  = *FPSCR;
       pc += 2;
}
STS_RESTORE(long  *FPSCR,*Rn)          /* STS.L FPSCR,@-Rn */
{
long  *tmp_address = *Rn - 4;
      if(store_long(FPSCR,tmp_address) != Address_Error)
      Rn  = tmp_address;
      pc += 2;
}
```

Exceptions:

Address Error



*1ÅFÉvÉãÉAÉbÉvÇ
µÇƒÇ≠ÇæÇ≥Ç¢ÅB
*2ÅFâΩÇ‡ê⁄ë±ÇµÇ
»Ç¢Ç≈Ç≠ÇæÇ≥Ç¢ÅB

FLOAT (Floating Point Convert from Integer): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FLOAT FPUL,FRn	(float)FPUL -> FRn	1111nnnn00101101	2	1	-

Description: Interprets the content of FPUL as an integer value and converts it to a floating point number value. And the result of this operation is written on the floating point register FRn.

Operation:

```
FLOAT(int *FPUL, float *FRn)  /* FLOAT FRn */
{
      clear_cause_VZ();
      *FRn  = (float) *FPUL;
       pc += 2;
}
```

Exceptions:

None

FMAC (Floating Point Multiply Accumulate): Floating Point Instruction

Format	Abstract	Code	Latency	Pitch	T bit
FMAC FR0,FRm,FRn	FR0*FRm+FRn -> FRn	1111nnnnmmmm1110	2	1	-

Description: Floating-point-arithmetically multiplies the contents of floating point registers FR0 and FRm. And the result of this operation is accumulated to floating point register FRn.

Operation:

```
FMAC(float *FR0,*FRm,*_FRn)    /* FMAC FR0,FRm,_FRn */
{
long    tmp_FPSCR;
float *tmp_FMUL = *FRm;
      FMUL(FR0,tmp_FMUL);
      pc -= 2;              /* correct pc                          */
      tmp_FPSCR = FPSCR;    /* save cause field for FR0*FRm   */
      FADD(tmp_FMUL,FRn);
      FPSCR |= tmp_FPSCR;   /* reflect cause field for FR0*FRm      */
}
```

FMAC Special Cases

FRn	FR0	FRm: +NORM	-NORM	+0	-0	+INF	-INF	qNaN	sNaN
NORM	NORM	MAC				INF			
	0					Invalid			
	+INF	+INF	-INF	Invalid		+INF	-INF		
	-INF	-INF	+INF			-INF	+INF		
+0	NORM	MAC				INF			
	0				+0	Invalid			
	+INF	+INF	-INF	Invalid		+INF	-INF		
	-INF	-INF	+INF			-INF	+INF		
-0	+NORM	MAC		+0	-0	+INF	-INF		
	-NORM			-0	+0	-INF	+INF		
	+0	+0	-0	+0	-0	Invalid			
	-0	-0	+0	-0	+0				
	+INF	+INF	-INF	Invalid		+INF	-INF		
	-INF	-INF	+INF			-INF	+INF		
+INF	+NORM	+INF					Invalid		
	-NORM						+INF		
	0					Invalid			
	+INF		Invalid			+INF			
	-INF	Invalid	+INF				+INF		
-INF	+NORM	-INF					-INF		
	-NORM								
	0								
	+INF	Invalid		Invalid			-INF		
	-INF	-INF				-INF	Invalid		
qNaN	0					Invalid			
	INF			Invalid					
	! sNaN								
~! NaN	qNaN							qNaN	
all types	sNaN								
sNaN	all types								Invalid

Denormalized value is treated as ZERO.

Exceptions:

Invalid operation

FMOV (Floating Point Move): Floating Point Instruction

	Format	Abstract	Code	Latency	Pitch	T bit
1	FMOV FRm, FRn	FRm -> FRn	1111nnnnmmmm1100	2	1	-
2	FMOV.S @Rm, FRn	(Rm)-> FRn	1111nnnnmmmm1000	2	1	-
3	FMOV.S FRm, @Rn	FRm ->(Rn)	1111nnnnmmmm1010	2	1	-
4	FMOV.S @Rm+,FRn	(Rm)-> FRn, Rm+=4	1111nnnnmmmm1001	2	1	-
5	FMOV.S FRm,@-Rn	Rn-=4, FRm->(Rn	1111nnnnmmmm1011	2	1	-
6	FMOV.S @(R0,Rm),FRn	(R0+Rm)-> FRn	1111nnnnmmmm0110	2	1	-
7	FMOV.S FRm,@(R0,Rm)	FRm->(R0+Rn)	1111nnnnmmmm0111	2	1	-

Description:

1. Moves the content of floating point register FRm to the floating point register FRn.
2. Loads the content of the memory location addressed by general register Rm. And the result of this operation is written on the floating point register FRn.
3. Stores the content of floating point register FRm into the memory location addressed by general register Rn.
4. Loads the content of the memory location addressed by general register Rm. And the result of this operation is written on the floating point register FRn. Upon successful completion, the value in Rm is incremented by 4.
5. Stores the content of floating point register FRm into the memory location addressed by general register Rn decremented by 4. Upon successful completion, the decremented value becomes the value of Rn.
6. Loads the content of the memory location addressed by general register Rm and R0. And the result of this operation is written on the floating point register FRn.
7. Stores the content of floating point register FRm into the memory location addressed by general register Rn and R0.

Operation:

```
FMOV(float *FRm,*FRn)    /* FMOV.S FRm,FRn */
{
      *FRn  = *FRm;
       pc += 2;
}
FMOV_LOAD(long *Rm,float *FRn)              /* FMOV @Rm,FRn */
{
      load_long(Rm,FRn);
      pc += 2;
}
FMOV_STORE(float *FRm,long *Rn)      /* FMOV.S FRm,@Rn */
{
      store_long(FRm,Rn);
      pc += 2;
}
FMOV_RESTORE(long *Rm,float *FRn)    /* FMOV.S @Rm+,FRn */
{
      if(load_long(Rm,FRn) != Address_Error)     *Rm += 4;
      pc += 2;
}
FMOV_SAVE(float *FRm,long *Rn)              /* FMOV.S FRm,@-Rn */
{
long  *tmp_address = *Rn - 4;
      if(store_long(FRm,tmp_address) != Address_Error)       Rn = tmp_address;
      pc += 2;
}
FMOV_LOAD_index(long *Rm, long *R0, float *FRn)        /* FMOV.S @(R0,Rm),FRn */
{     load_long(&(*Rm+*R0) ,FRn) ;
```

```
        pc += 2;
}
FMOV_STORE_index(float *FRm,long *R0, long *Rn) /* FMOV.S FRm,@(R0,Rn) */
{
        store_long(FRm,&(*Rn+*R0));
        pc += 2;
}
```

Exceptions:

Address Error

2. CPU/FPU Interface Signals

Name	Phase	Width	Direction
s2_d	ph2	32	S bus to FPU
c2_sbrdy	ph2	1	CCN to FPU
t2_tlbmiserr	ph2	1	MMU to FPU
u2_abrk	ph2	1	UBC to FPU
u2_brktyp	ph2	3	UBC to FPU
s2_fstall	ph2	1	CPU to FPU
s2_dfcirsel	ph2	4 (1-hot)	CPU to FPU
s1_dfcirtseld	ph1	1	CPU to FPU
s2_irthihl	ph2	1	CPU to FPU
s2_irthill	ph2	1	CPU to FPU
s2_fbypass	ph2	4	CPU to FPU
s2_fcancel1	ph2	1	CPU to FPU
s1_fcancel2	ph1	1	
s1_iinvalid	ph1	1	CPU to FPU
f2_tbit	ph2 (E1)	1	FPU to CPU
f2_busy	ph2 (D)	1	FPU to CPU
f2_exc	ph2 (E1)	1	FPU to CPU

Notes:
1. s2_ffreeze may be necessary. (Any particular necessity has not yet been identified)
2. This list only includes the CPU/CCN/FPU signals. It does not include signals such as reset, l2_lrdy, etc.
3. Due to lack of detailed information regarding SH3 CPU logic, we are not sure if all the above signals are reall necessary. Redundancy of signals will be eliminated as the simulation and validation proceeds.

2.1 Signal Descriptions

2.1.1 s2_d — Carries the S-Bus data between the CPU, CCN, MAC and FPU. The FPU will monitor it during instruction fetches, floating point loads and LDS instructions that write to the FPUL register. It will put data on the bus during floating point stores and STS instructions that read the FPUL register.

2.1.2 c2_sbrdy — Indicates when the data on s2_d is ready. It is also used to indicate cache misses.

2.1.3 t2_tlbmiserr — Indicates that some form of a TLB exception has occurred. This signal will tell the FPU when to cancel floating point operations that are in progress.

2.1.4 u2_abrk — Indicates that an address break has occurred.

2.1.5 u2_brktyp — Indicates the type of the address break.

2.1.6 s2_fstall — indicates a NOP needs to be introduced at the E1-stage of the FPU pipe.

2.1.7 s2_dfcirsel — 4 bit one-hot signals from CPU to control I-Fetch databath 4 input Mux.

2.1.8 s1_dfcirtseld — signal to I-fetch dtapath to select the high or low of the I-word to be loaded in the instruction buffer.

2.1.9 s2_irthihl — selection for the high order bytes of the instruction on the s2_d bus.

2.1.10 s2_irthill — selection for the low order bytes of the instruction on the s2_d bus.

2.1.11 s2_fbypass — Tells the FPU which data to forward to each instruction. (see section 3.8)

2.1.12 s1_fcancel2 — Tells the FPU to cancel the instructions in its I, D, E1 and E2 stages.

2.1.13 s1_iinvalid — Tells the FPU to ignore the instructions on the s2_d bus.

2.1.14	f2_tbit	Condition code generated by the floating point compare instructions.
2.1.15	f2_busy	Indicates that the FPU is executing a floating point divide or square root.
2.1.16	f2_exc	Indicates that there was an FPU exception.
2.1.17	s2_fcancel1	Tells the FPU to cancel the instruction in I and D stages

2.2 Signals that should be synthesized at the FPU end

Some signal names are used in this documentation that are not directly available as interface signals. We assume that these signals will be formed at the FPU end from the available interface signals.

2.2.1 s2_ifetch OR (s2_irthihl , s2_irthill) , signal indicates instruction fetch is in progress.

3.8 FPU Bypass Cases

To improve its instruction throughput, the FPU supports register forwarding. Six bypass paths are used.

1. S-bus (data bus 222) to E1 input (Fm)
2. S-bus to E1 input (Fn)
3. S-bus to E2 input
4. E2 output to E2 input
5. E2 output to E1 input (Fn)
6. E2 output to E1 input (Fm)
7. E2 output to Fmac input (F0)
8. S-bus to Fmac input (F0)

Four bit signal s2_fbypass from the CPU controls bypass at the FPU end. The encoding is as follows:

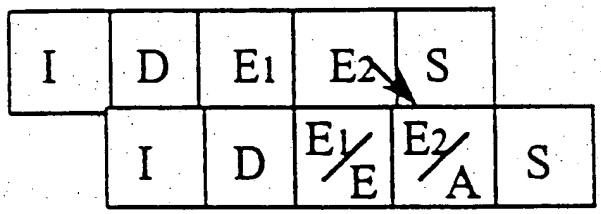
s2_fbypass[4]:bypass to Fmac input (F0)
s2_fbypass[3]:bypass to E2 input
s2_fbypass[2]:bypass to E1 input (Fn)
s2_fbypass[1]:bypass to E1 input (Fm)

s2_fbypass[0] = 1:bypass from S-bus
s2_fbypass[0] = 0:bypass from E2

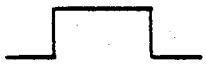
3.8.1 Bypass from E_2 output to E_2 input

FADD FR3,FR1

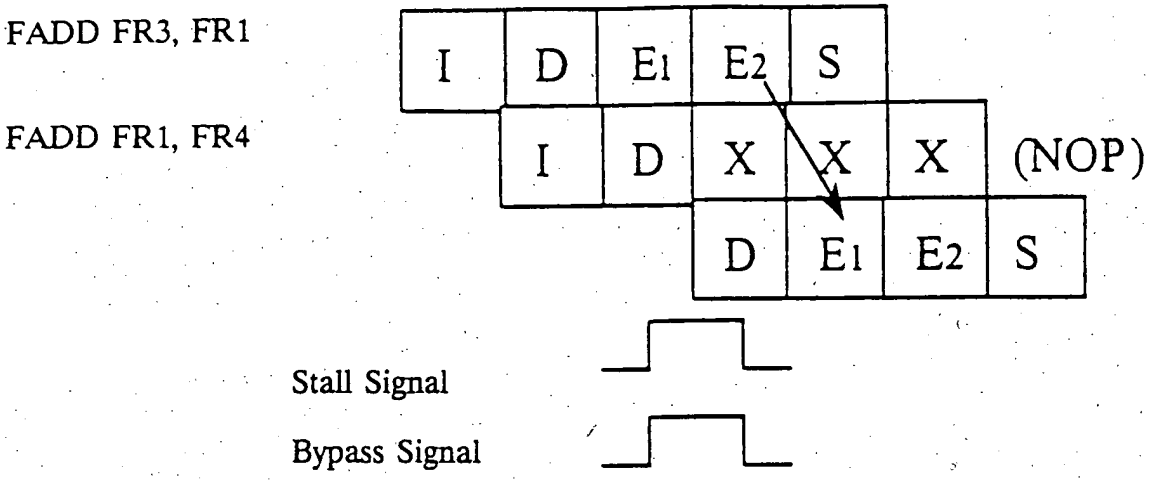
FMOV.SFRI.@Rn



Bypass Signal



3.8.2 Bypass from E_2 output to Rn or Rm input of E_1



3.9.1 Bypass from data bus 222 to Rn or Rm input of E_1